

# Capítulo 3: Capa Transporte - III

## ELO322: Redes de Computadores Agustín J. González

Este material está basado en:

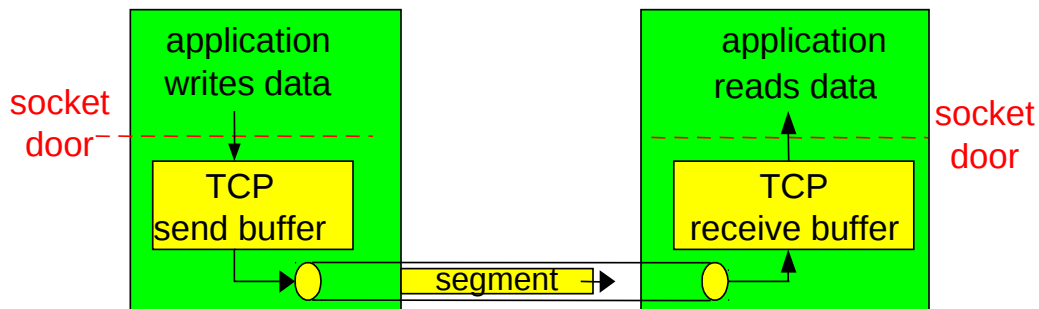
- Material de apoyo al texto *Computer Networking: A Top Down Approach Featuring the Internet 3rd* edition. Jim Kurose, Keith Ross Addison-Wesley, 2004.

# Capítulo 3: Continuación

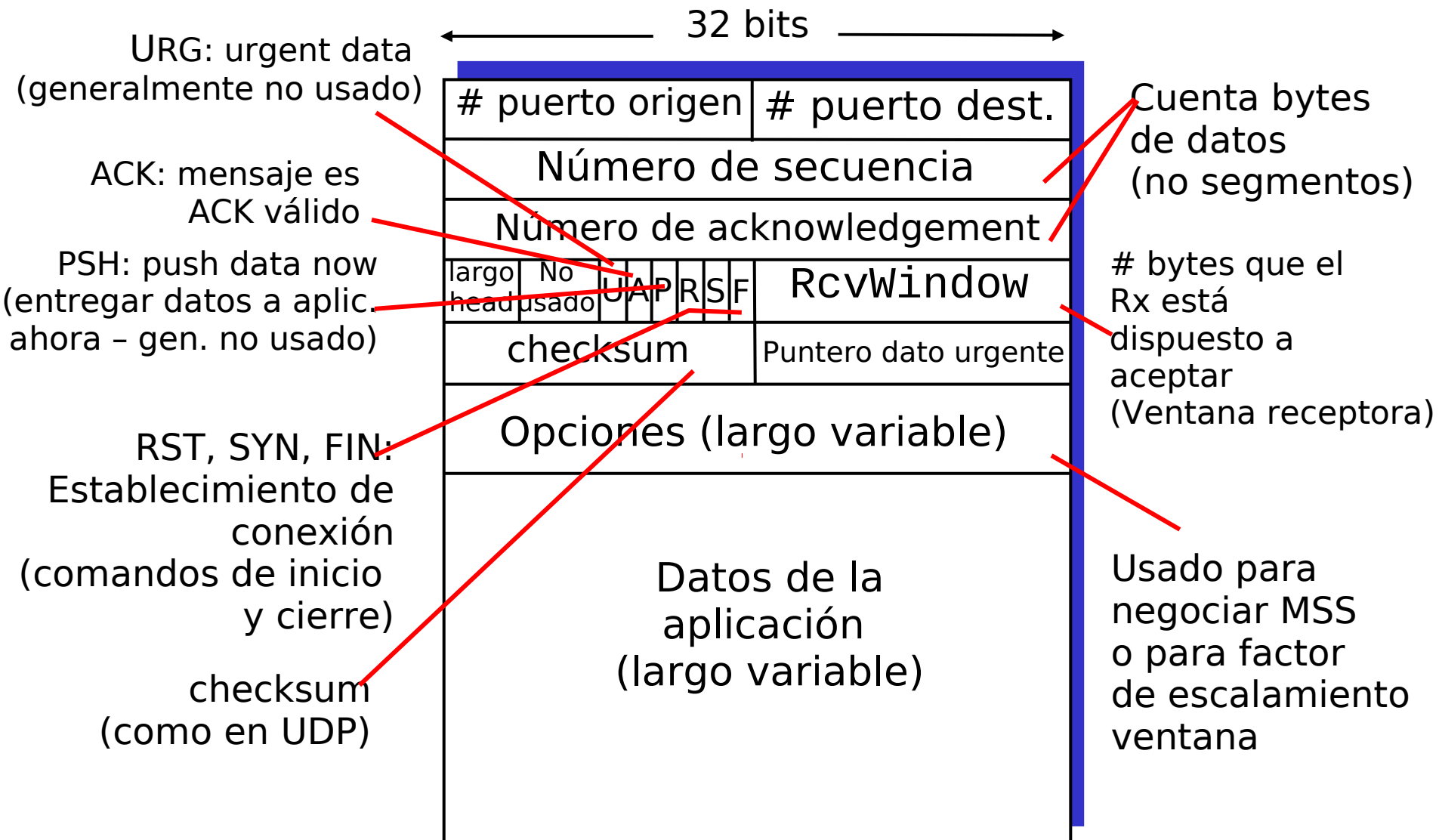
- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - Transferencia confiable de datos
  - Control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP

# TCP: Generalidades RFCs: 793, 1122, 1323, 2018, 2581

- Es una comunicación Punto-a-punto:
  - Un Tx y un Rx
- **Confiable flujo de bytes en orden:**
  - No hay “límites del inicio/término de mensaje”
- **Usa pipeline:**
  - El tamaño de la ventana TCP es definido por el control de congestión y control de flujo
- **Usa buffer en Tx & Rx**
- **Transferencia full duplex (dos sentidos):**
  - Flujo de datos bi-direccionales en la misma conexión
  - MSS: maximum segment size (depende del MTU)
- **Orientado a la conexión:**
  - Handshaking (intercambio de mensajes de control) inicializa al Tx y Rx antes del intercambio de datos
- Tiene control de **flujo:**
  - Tx no sobrecargará al Rx
- También tiene control de **congestión**



# Estructura de un segmento TCP



# Ejemplo: Uso de # de sec. y ACKs en Telnet (Aplicación sobre TCP)

## # de Sec.:

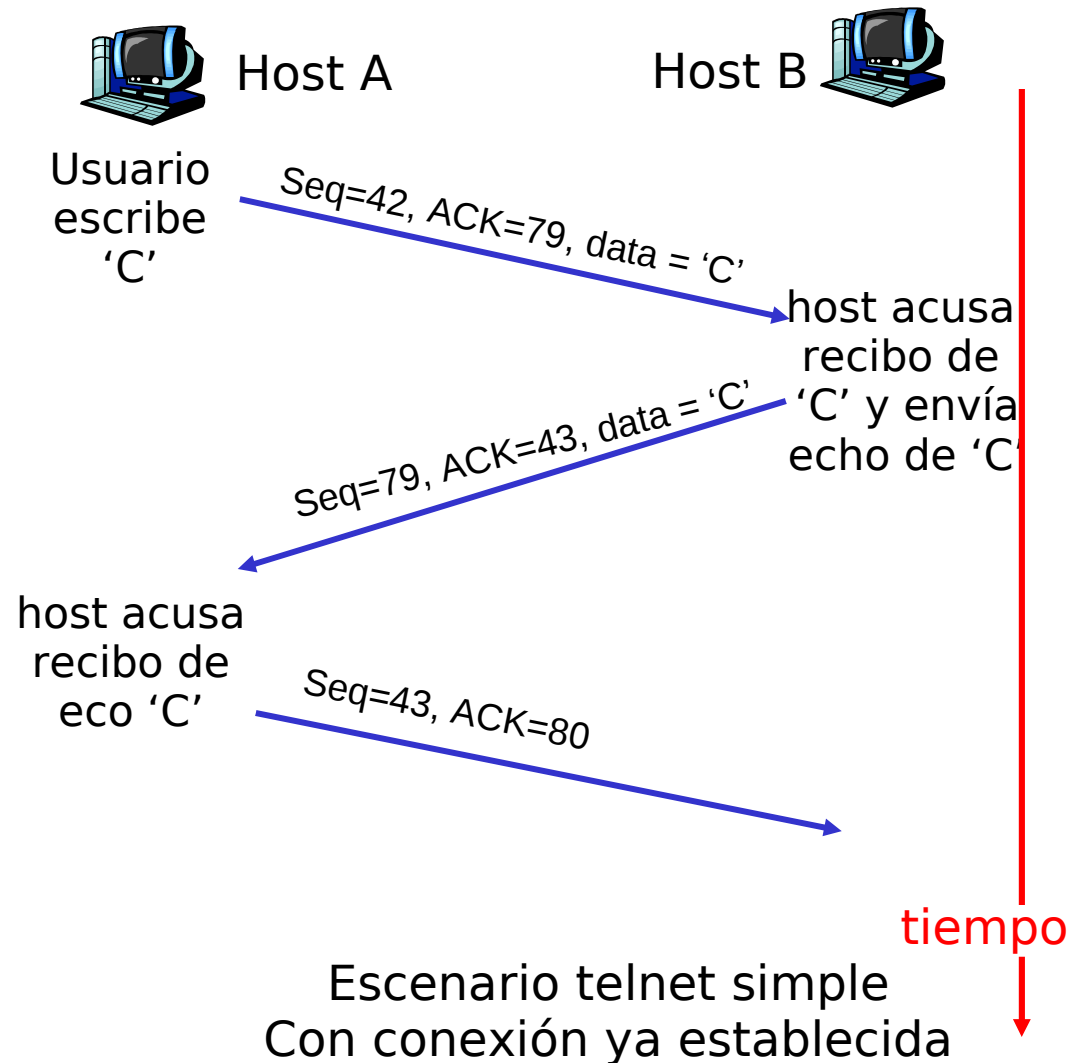
- ▢ “número” del byte dentro del flujo correspondiente al primer byte del segmento de datos

## ACKs:

- ▢ # sec. del próximo byte esperado desde el otro lado
- ▢ ACK es acumulativo

**Q:** ¿Cómo el receptor maneja segmentos fuera de orden?

- ▢ la especificación de TCP lo deja a criterio del implementador



# Round Trip Time y Timeout en TCP

Q: ¿Cómo fijar valor de timeout en TCP?

- Mayor que RTT
  - pero RTT varía
- Muy corto: timeout prematuro
  - Retransmisiones innecesarias
- Muy largo: lenta reacción a pérdidas de segmentos

Q: ¿Cómo estimar el RTT?

- **SampleRTT**: mide tiempo desde tx del segmento hasta recibo de ACK
  - Ignora retransmisiones
- **SampleRTT** varía, hay que suavizar el RTT estimado
  - Promediar varias medidas recientes, no considerar sólo el último **SampleRTT**

# Round Trip Time y Timeout en TCP

$$\text{EstimatedRTT}_i = (1 - \alpha) * \text{EstimatedRTT}_{i-1} + \alpha * \text{SampleRTT}_i$$

- Promedio móvil ponderado exponencial
- Influencia de las muestras pasadas decrece exponencialmente rápido

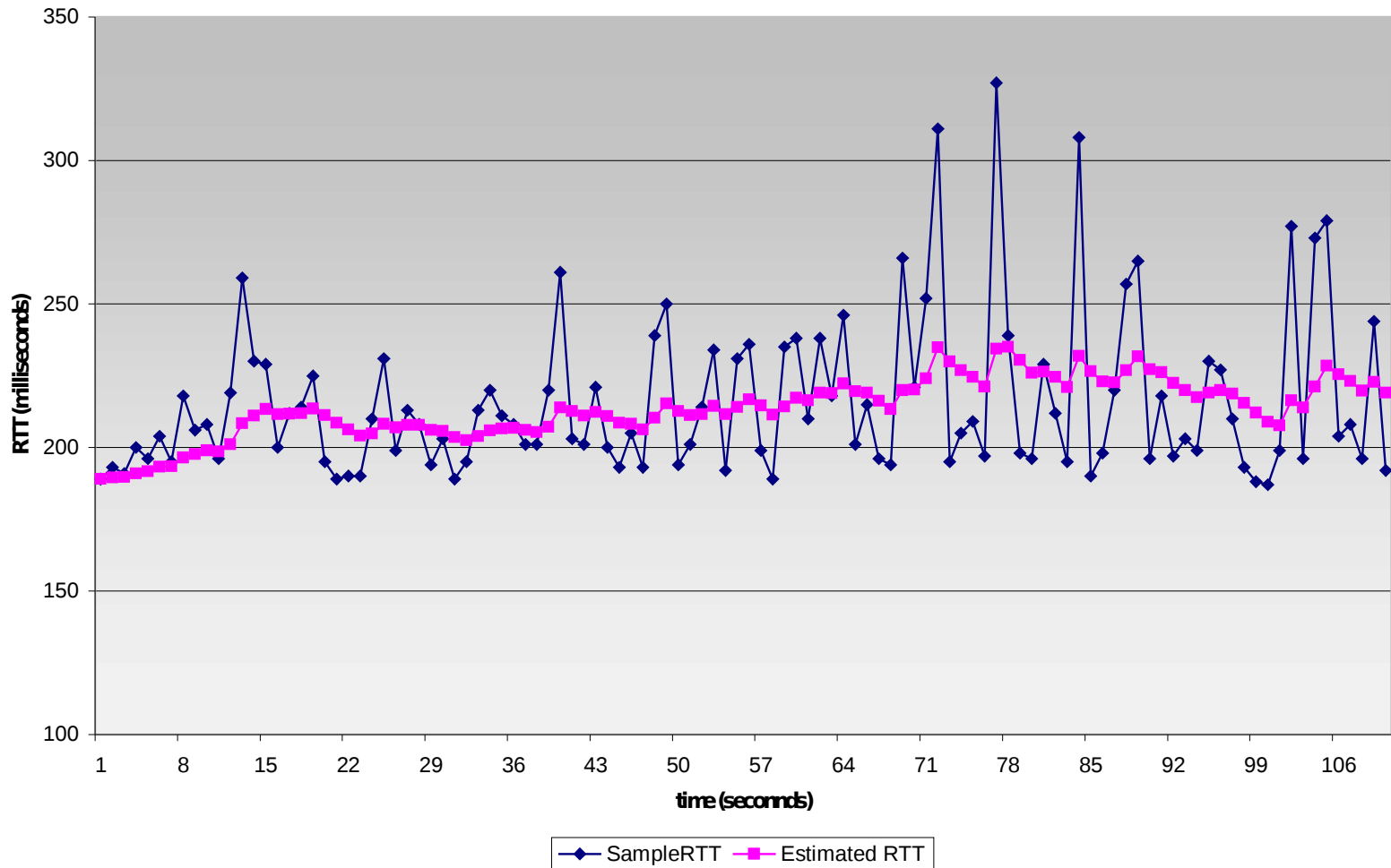
Ejercicio: anote  $\text{EstimatedRTT}_i$  en función  $\text{SampleRTT}_1.. \text{SampleRTT}_i$

- Valor típico:  $\alpha = 0.125 = (1/8) = 3$  right shifts en binario.

(aunque hoy un right shift tiene costo similar a una multiplicación)

# Ejemplo de estimación de RTT:

RTT: `gaia.cs.umass.edu` to `fantasia.eurecom.fr`





# Timeout en TCP

- Timeout usa **EstimatedRTT** más un “margen de seguridad”
  - Si hay gran variación en **EstimatedRTT** => usar gran margen
- Primero estimamos cuánto se desvía el **SampleRTT** del **EstimatedRTT**:

$$\text{DevRTT}_i = (1 - \beta) * \text{DevRTT}_{i-1} + \beta * |\text{SampleRTT}_i - \text{EstimatedRTT}_i|$$

(típicamente,  $\beta = 0.25$ )

Entonces TCP fija el timeout como:

$$\text{TimeoutInterval}_i = \text{EstimatedRTT}_i + 4 * \text{DevRTT}_i$$

# Capítulo 3: Continuación

- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - **Transferencia confiable de datos**
  - Control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP

# Transferencia confiable de datos en TCP

- TCP crea un servicio de transferencia confiable sobre el servicio no confiable de IP
- Usa segmentos en un pipeline
- ACKs acumulativos
- TCP usa un timer de retransmisión único
- Retransmisiones son activadas por:
  - Eventos de timeout
  - ACKs duplicados
- Inicialmente consideremos un Tx TCP simplificado:
  - Ignora **ACKs duplicados**
  - Ignora **control de flujo y control de congestión**

# Eventos del transmisor en TCP:

## Datos recibidos desde aplicación:

- Crea segmento con # de sec.
- # sec es el número dentro del flujo para el primer byte del segmento
- Inicia timer si no está ya corriendo (asociar el timer al segmento más viejo sin acuse de recibo)
- Tiempo de expiración: TimeoutInterval

## Timeout:

- Retransmitir el segmento que causó el timeout
- Típicamente el intervalo del timeout se duplica en retransmisiones. ¿Por qué?
- Re-iniciar el timer

## Recepción de ACK:

- Si es el ACK de un segmento previo sin acuse de recibo
  - Actualizar estado sobre acuses recibidos
  - Iniciar timer si hay segmentos sin acuses de recibo.

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
```

```
loop (forever) {
  switch(event)
```

```
  event: datos recibidos desde la aplicación
    Crear segmento TCP con número de sec. NextSeqNum
    if (timer actualmente no está corriendo)
      iniciar timer
    pasar segmento a IP (capa red)
    NextSeqNum = NextSeqNum + length(data)
    break;
```

```
  event: timeout del timer
    retransmitir segmento con menor # de sec. sin acuse
    iniciar timer
    break;
```

```
  event: recepción de ACK con campo ACK de valor x
    if (x > SendBase) {
      SendBase = x
      if (hay segmentos sin acuse de recibo aún)
        iniciar timer
      else detener timer
    }
```

```
} /* end of loop forever */
```

## Tx TCP (simplificado)

### Comentarios:

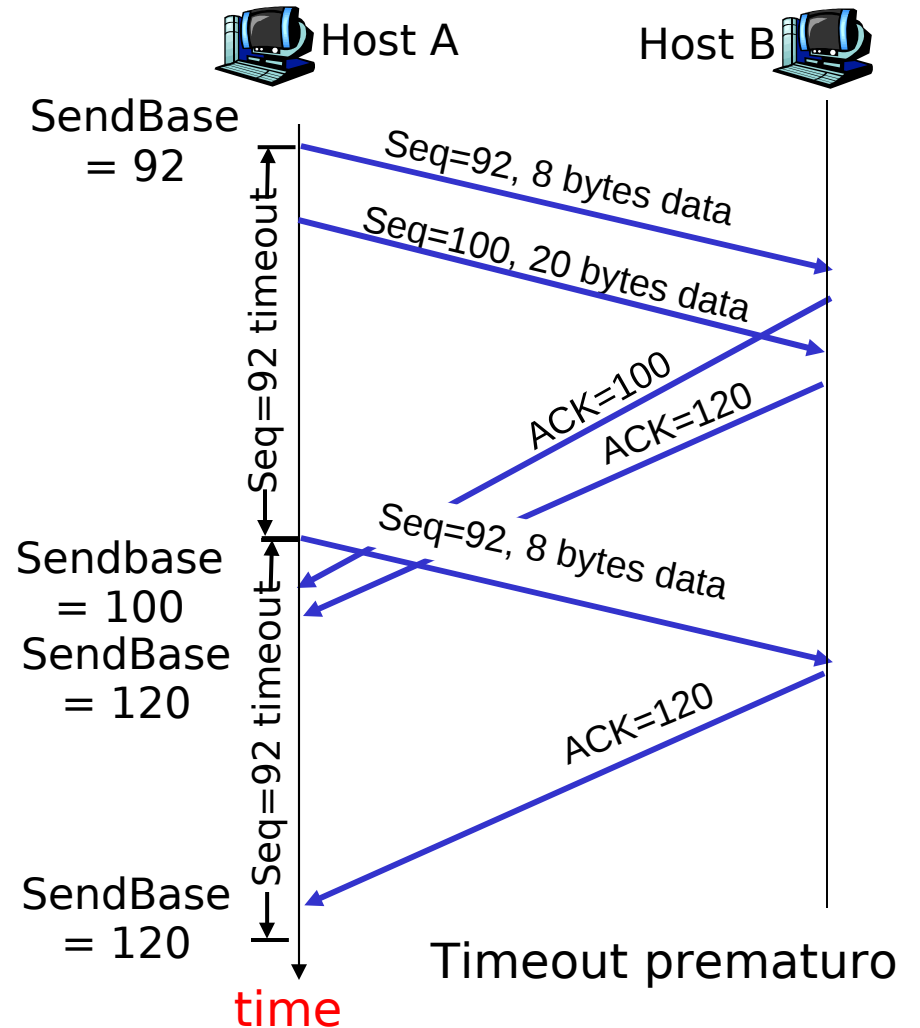
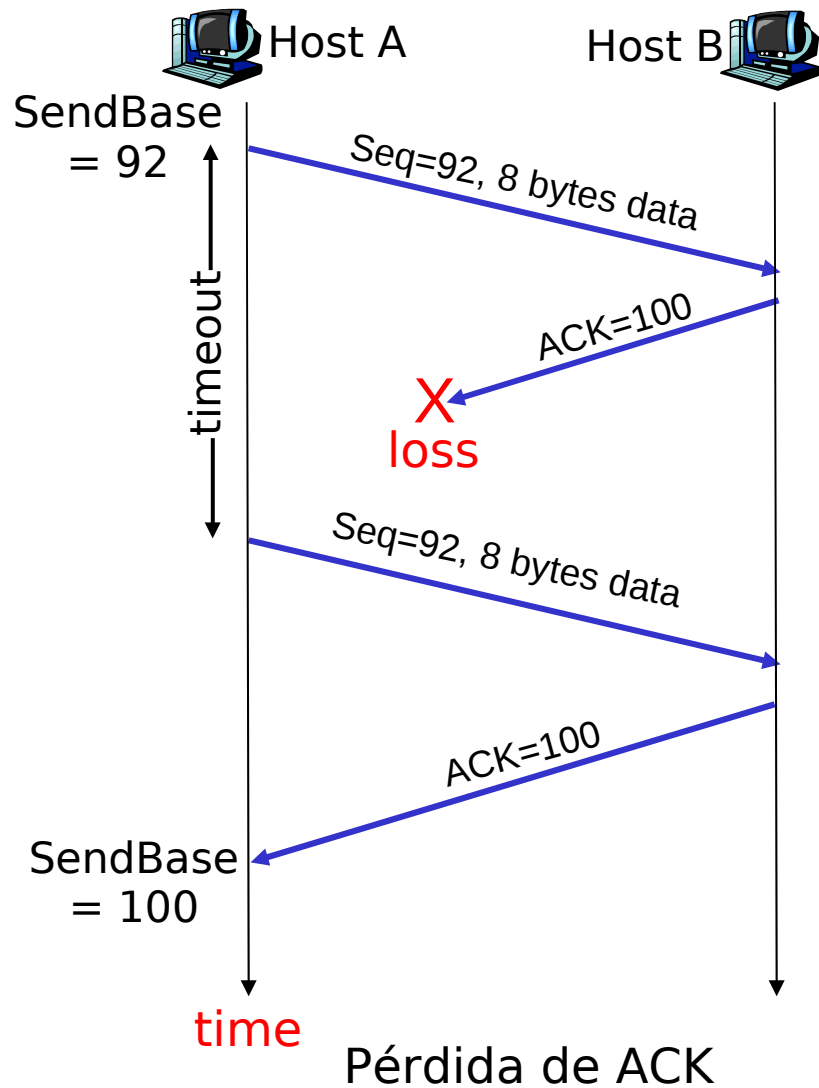
- SendBase: Byte más antiguo sin ACK
- SendBase-1: último Byte con ACK recibido

### Ejemplo:

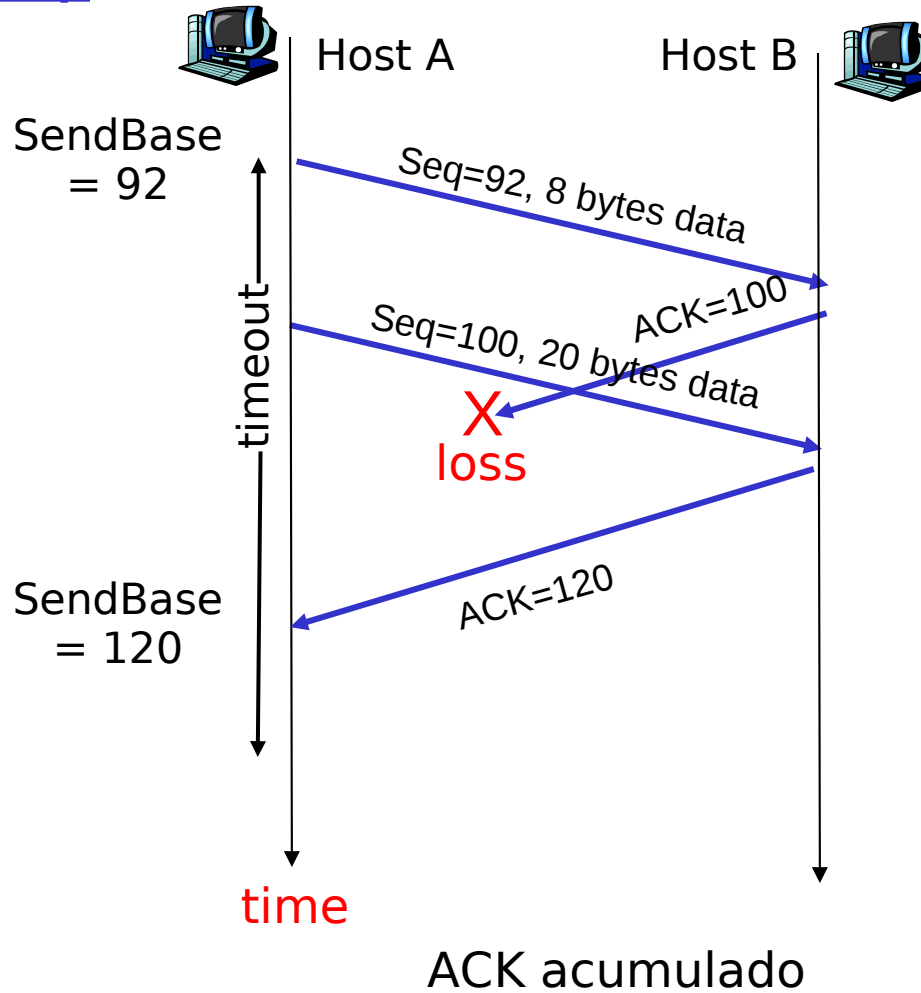
- SendBase = 71 y se recibe ACK con  $x = 72$
- El Rx quiere nuevos bytes con  $\text{seq} = 72$
- Como  $x > \text{SendBase}$ , llegó acuse de recibo de dato ( $\text{seq} = 71$ )

Notar que evento de timeout envía sólo el paquete más antiguo.

# TCP: escenarios de retransmisión



# TCP escenarios de retransmisión (más)



# Generación de ACK en TCP [RFC 1122, RFC 2581]

Notar efecto en RTT

## Evento en Receptor

## TCP acción del receptor

Llegada de segmento en orden con # sec. esperado. Ya se envió el ACK de todo lo previo.

ACK retardado. Espera hasta 500ms por próximo segmento. Si no llega otro segmento enviar ACK

Llegada de segmento en orden con # sec. esperado. Algún segmento tiene ACK pendiente

Enviar inmediatamente un ACK acumulado se da acuse así a ambos segmentos en orden.

Llegada de segmento fuera de orden con # sec. mayor al esperado. Se detecta un vacío.

Enviar inmediatamente un ACK duplicado, indicando # sec. del próximo byte esperado

Llegada de segmento que llena el vacío parcialmente o completamente

Enviar inmediatamente un ACK si es que segmento se ubica al inicio del vacío de segmentos recibidos



# TCP: Retransmisiones Rápidas

- Periodo de Time-out es a menudo largo:
  - Largo retardo antes de re-envío de paquetes perdidos
- Se detecta paquetes perdidos vía ACKs duplicados.
  - Tx a menudo envía muchos segmentos seguidos
  - Si un segmento es perdido, probablemente habrá muchos ACKs duplicados.
- Si Tx recibe 3 ACKs de un mismo dato, éste supone que el segmento después de este dato se perdió:
  - Retransmisiones rápidas: reenviar el segmento antes que el timer expire.

# TCP: Algoritmo de Retransmisión Rápida

```
event: Llega ACK, con campo ACK de valor x
  if (x > SendBase) {
    SendBase = x
    if (hay segmentos sin acuse de recibo aún)
      iniciar timer
    esle detener timer
  }
  else { // x == SendBase
    incrementar cuenta de ACKs de x duplicados
    if (cuenta de ACKs de x duplicados == 3) {
      re-enviar segmento con # de secuencia x
      iniciar timer
    }
  }
```

ACK duplicado de un segmento con ACK recibido

Retransmisión rápida

# TCP: Timeout

## Doblando el tiempo del timeout

- Algunas modificaciones en muchas implementaciones de TCP:
- La primera concierne al largo de tiempo del intervalo del timeout después que el timer expira
  - En esta modificación cuando ocurre un timeout, TCP retransmite el segmento sin ACK con menor número de secuencia pero para cada retransmisión TCP **duplica** el valor previo de TimeoutInterval
  - De esta forma los intervalos crecen exponencialmente después de cada retransmisión sucesiva.
- La segunda es si se recibe un ACK entonces se **recalcula** TimeoutInterval usando EstimatedRTT y DevRTT
- Esta es una forma limitada de **control de congestión**

# Capítulo 3: Continuación

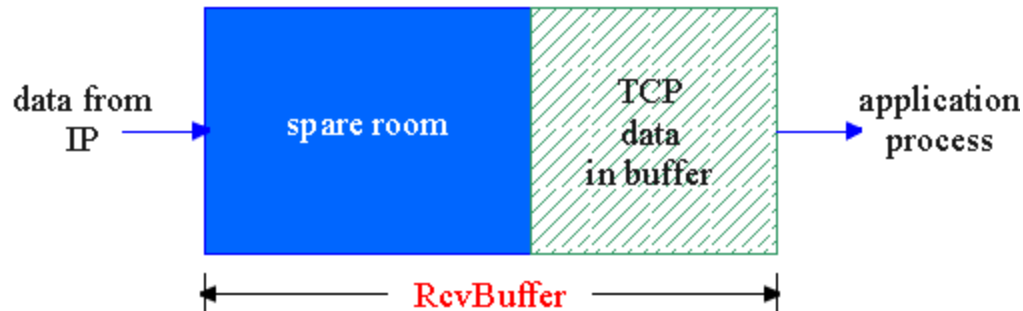
- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - Transferencia confiable de datos
  - **Control de flujo**
  - Administración de conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP

# Control de flujo en TCP

□ Hemos visto cómo TCP asegura **confiabilidad** en la transferencia, ahora veremos cómo consigue controlar el **flujo de datos**.

□ El lado receptor (Rx) de TCP tiene un buffer receptor de datos:

← RevWindow →



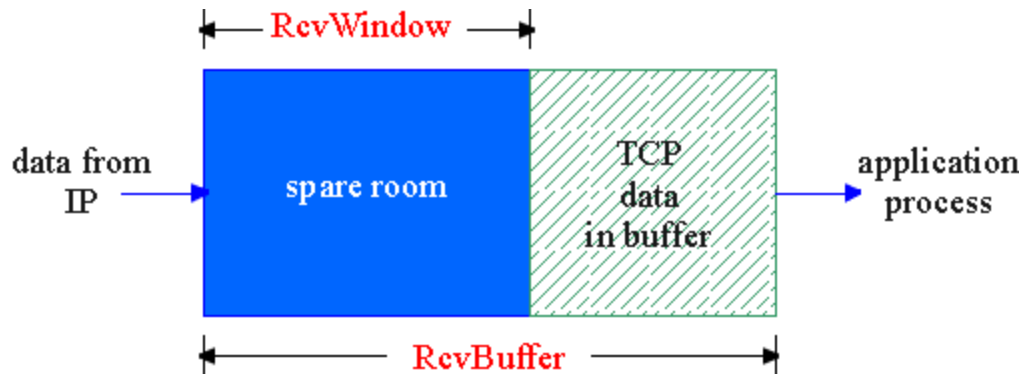
□ El proceso aplicación puede ser lento en la lectura del buffer (capa transporte).

## Control de flujo

Tx no debe sobrecargar el buffer del receptor por transmitir demasiado rápido

□ La idea es hacer coincidir la tasa de transmisión con la tasa de lectura de la aplicación.

# Control de flujo en TCP: Cómo trabaja



(supongamos que receptor descarta segmentos fuera de orden)

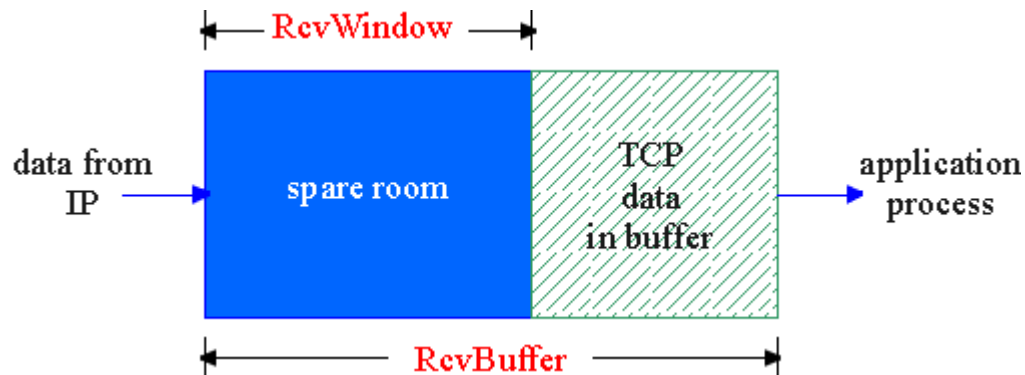
- Espacio libre en buffer

$$\text{RcvWindow} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

- Rx comunica el espacio libre a través del valor de **RcvWindow** en los segmentos
- Así el receptor limita datos en tránsito (sin ACK) a **RcvWindow** (Tx debe respetar el no envío de más datos que **RcvWindow**)
- Esto garantiza que el buffer del Rx no se rebalse (overflow)

# Control de flujo en TCP: Cómo trabaja

- El Transmisor debe tomar en cuenta los segmentos en tránsito
- Luego el número de bytes que el Tx puede enviar es en general menor que el anunciado por la RevWindows.
- ¿Cuál es la expresión para el número de Bytes posibles de enviar sin colapsar al receptor?



# Capítulo 3: Continuación

- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - Transferencia confiable de datos
  - Control de flujo
  - **Administración de conexión**
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP



# Administración de Conexión en TCP

**Recordar:** Transmisor y receptor TCP establecen una “conexión” antes de intercambiar segmentos de datos

- TCP inicializa variables:
  - # de secuencia
  - buffers, información de control de flujo (e.g. **RcvWindow**)

- *cliente:* Iniciación de conexión

```
Socket clientSocket = new  
Socket("hostname", "port  
number");
```

- *server:* contactado por cliente

```
Socket connectionSocket =  
welcomeSocket.accept();
```

## Saludo de manos de tres vías (Three way handshake):

**Paso 1:** host cliente envía segmento TCP SYN al servidor

- Especifica # secuencia inicial
- no data

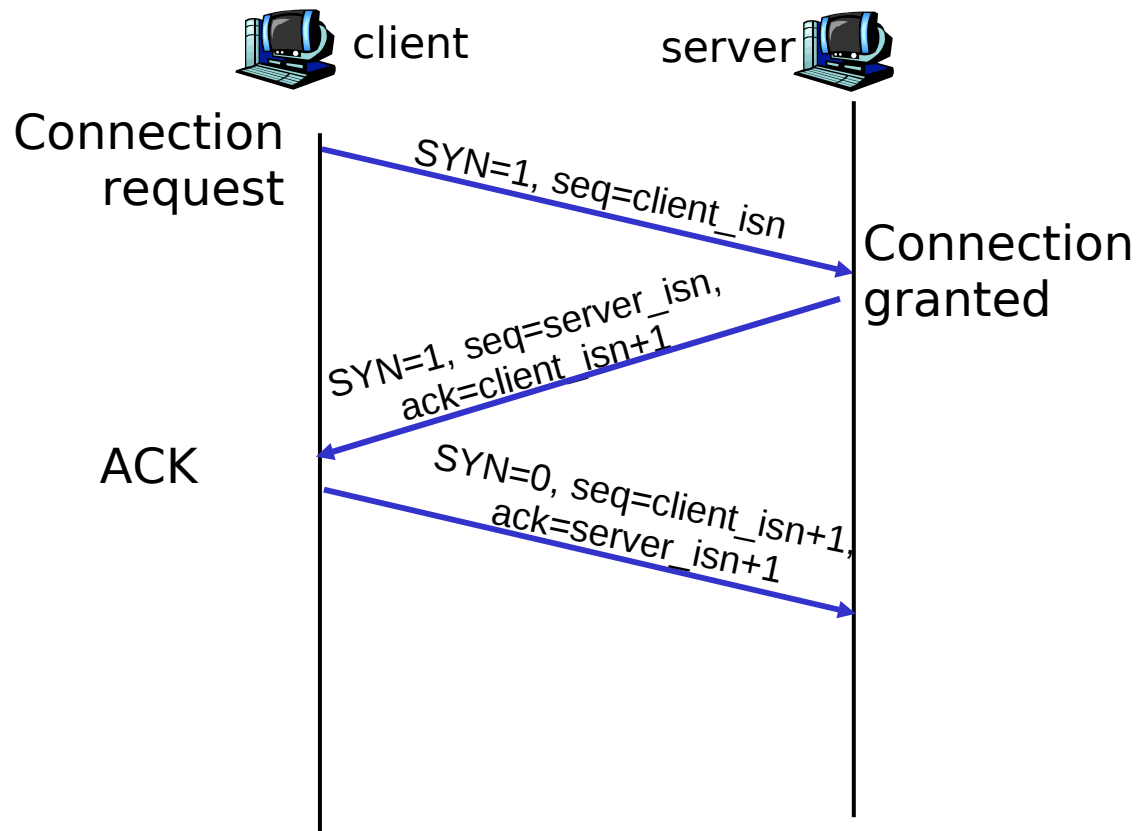
**Paso 2:** host servidor recibe SYN, responde con segmento SYN & ACK

- Servidor ubica buffers
- Especifica # secuencia inicial

**Paso 3:** cliente recibe SYN & ACK, responde con segmento ACK, el cual podría contener datos.

# Administración de Conexión en TCP (cont.)

## □ Establecimiento de conexión



# Administración de la conexión TCP (cont.)

## Cerrando una conexión:

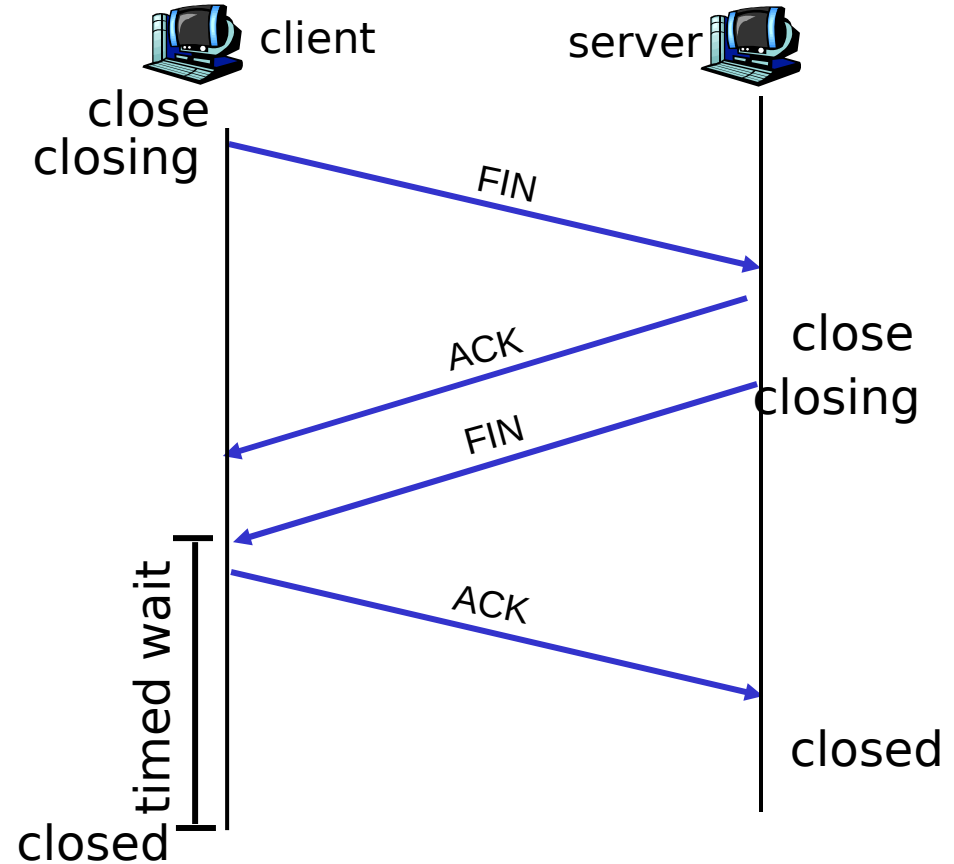
Cliente cierra socket:

```
clientSocket.close();
```

Puede ser iniciado por el servidor, aquí suponemos lo inicia el cliente.

**Paso 1:** host **cliente** envía segmento TCP FIN al servidor

**Paso 2:** **servidor** recibe FIN, responde con ACK. Ante un cierre de conexión de la aplicación y envía FIN.



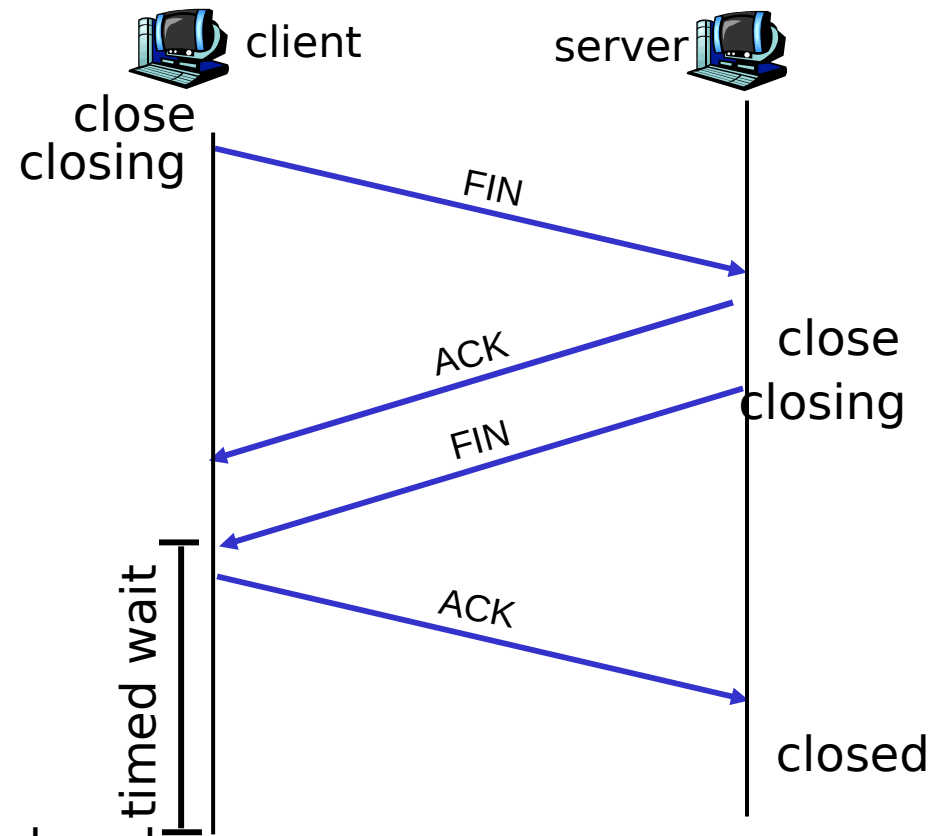
# Administración de la conexión TCP (cont.)

**Paso 3:** cliente recibe FIN, responde con ACK.

- ▢ Entra en “tiempo de espera” – responderá con ACK a FINs recibidos

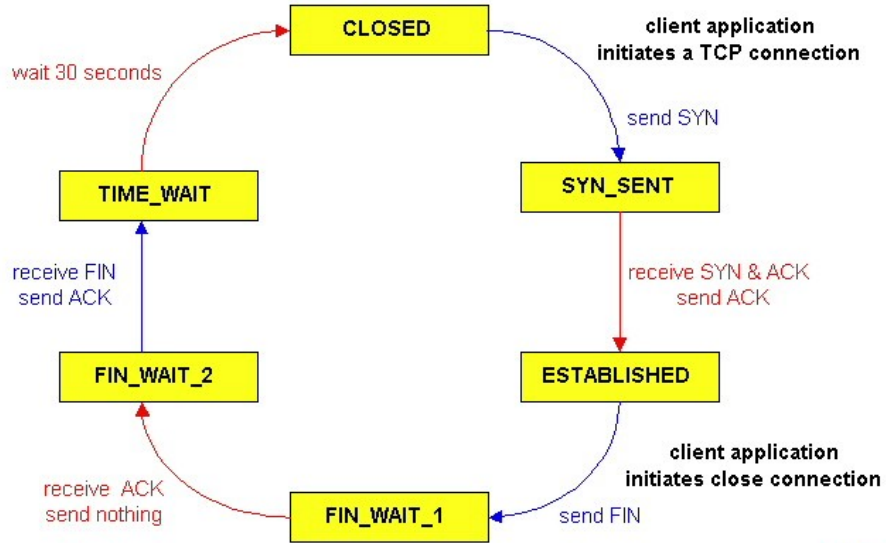
**Paso 4:** servidor, recibe ACK. Pasa a conexión cerrada.

**Nota:** Con pequeña modificación se puede manejar FINs simultáneos. closed



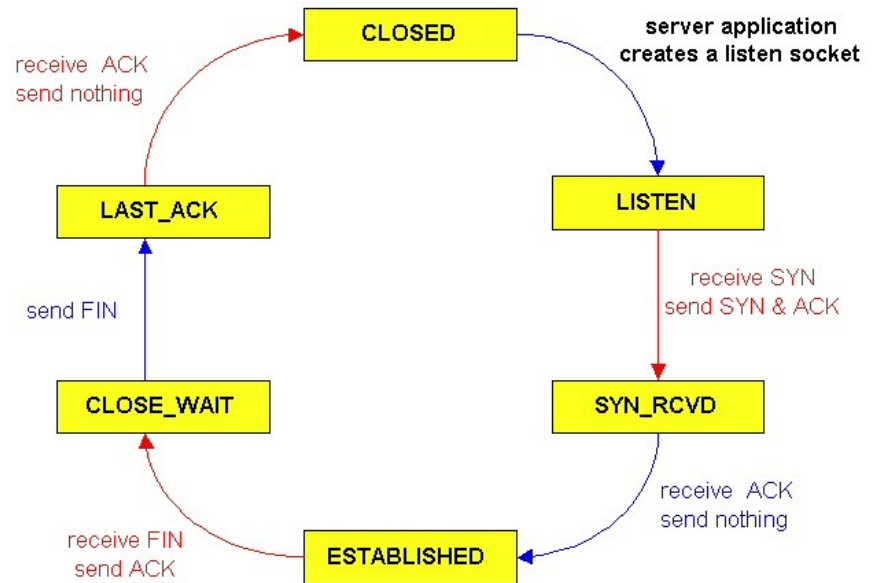
Cualquiera de los dos puede iniciar el cierre

# Administración de la Conexión TCP (cont)



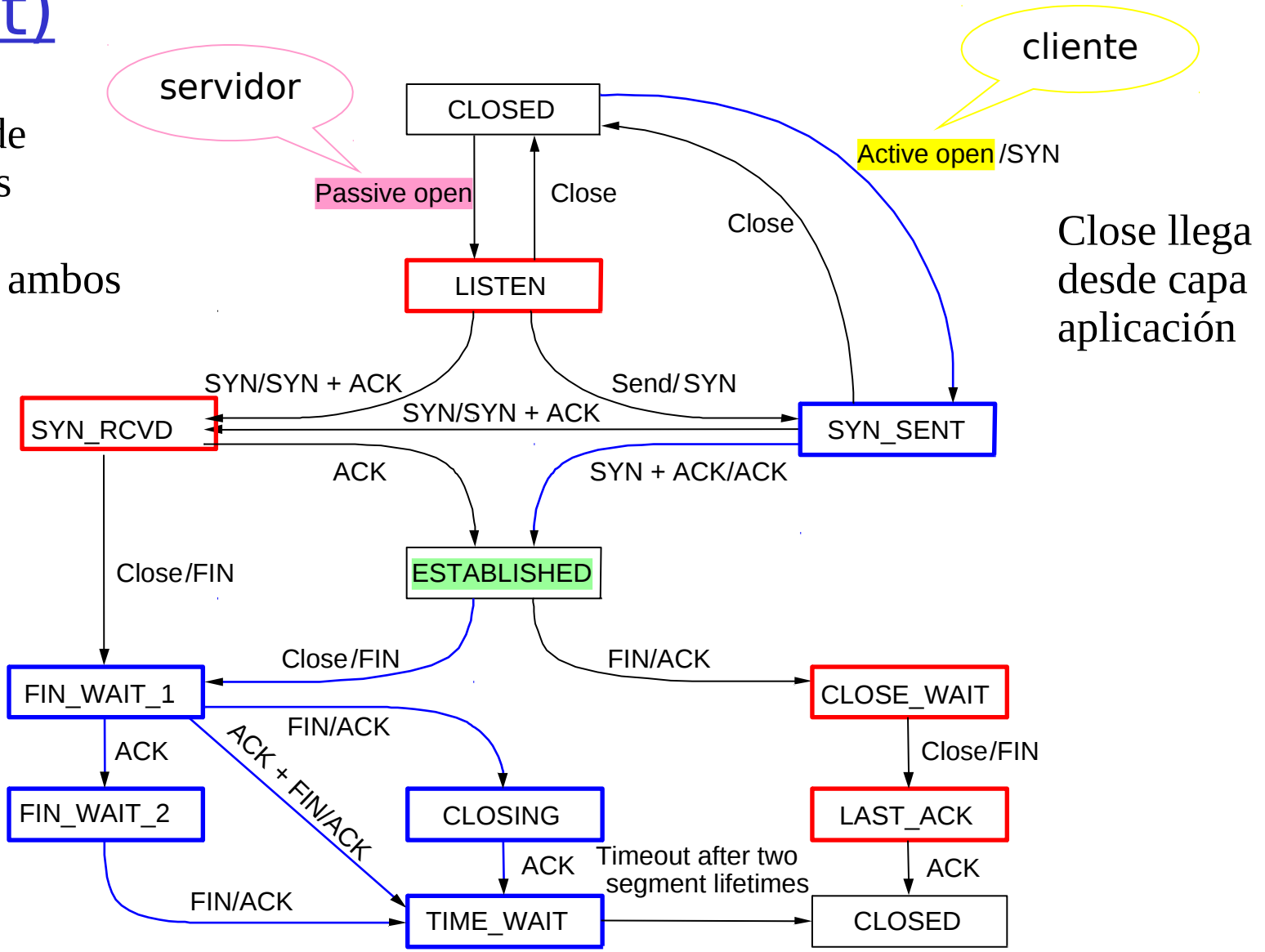
Ciclo de vida del cliente TCP

## Ciclo de vida del servidor TCP



# Administración de la Conexión TCP (cont)

Diagrama de estados más completo e incluyendo ambos casos



# Capítulo 3: Continuación

- 3.1 Servicios de la capa transporte
- 3.2 Multiplexing y demultiplexing
- 3.3 Transporte sin conexión: UDP
- 3.4 Principios de transferencia confiable de datos
- 3.5 Transporte orientado a la conexión: TCP
  - Estructura de un segmento
  - Transferencia confiable de datos
  - Control de flujo
  - Administración de conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión en TCP